APL-ication, held at UKC gave me the incentive to put pen to paper - 'fraid, until I had to chase up the address of Quote Quad, had not realised that you were the Editor.

On the way, used the new campus on line library catalogue - found that we do take Quote Quad - filed directly to the back periodical stacks !

Hope the following text is some use. There are not many Apl characters, so I have been lazy. I hope to have the chance to complete a consistent set of 8-bit fonts and translators, soon.

Suggest the following sequence :

```
uudecode apl_pipe.uue; arc xh apl_pipe; font_sw apl2.fed;

1stWordPlus  :
             : print apl_pipe.let
             : print apl_pipe.doc
             : open  syntax8b.doc     :(this is p2 of apl_pipe)
                                      :ScreenDump the equations
```

As to what it is, the following summarises :

(it is actually so straightforward,
that I thought it must be widely used)

A Pipe has two ends ...

Using Apl in a multiprocess/multiprocessor environment.

A proposal for a flexible but easy to use syntax.

I offer for general consideration a device that allows data to be piped out of Apl, through (a series of) shell commands, and back into Apl.

It works by connecting both ends of a pipeline of shell commands to an Apl statement.

The commands may be executed in the same processor, or remotely, with no change in syntax.

J.B. Webber. Rm. 15, Physics Lab.,
University of Kent, Canterbury.
Kent. CT2 7NR jbww@ukc.ac.uk.

Rm. 15, Physics Lab,
University of Kent,
Canterbury, Kent,
U.K., CT2 7NR

A.P.L. Quote Quad,
The Editor,
Prof. L.J. Dickey,
Dept. of Pure Mathematics,
University of Waterloo,
Waterloo, Ontario,
Canada. N2L 3G1

30st September, 1988

Dear Sir,

I enclose some notes on a proposed mechanism for an extension to Apl for use in multiprocess/multiprocessor systems, in the hopes that you may see fit to publish them.

I have made use of Apl for a number of years, but have not had the chance to follow all the developments in the field. The Apl that I have mainly used has been a version that runs under Unix, although I now increasingly use MicroAPl's Apl.68000.

Limitations that I found in the standard Unix Apl, and the need to access pre-compiled and C-shell commands led me to write into this Apl the ***pipe*** mechanism that I describe on the following pages.

I must say that I assumed that most other more current versions would include such a mechanism, as it seemed a particularly clean and straight forward interface.

The recent APL-ication meeting was held at Kent University, and I made use of the opportunity to examine some of the current versions of Apl being demonstrated. I had a word with some of the people on the stands concerning piping (particularly John Scholes of Dyalog), and close interest was expressed.

Thus I gather that such a mechanism is at the least not widely used. I do find it hard to believe it is original (though may have been when I first implemented it). If I am wrong, and this is a well known method, please forgive me for wasting your time.

However, with the increasing interest in making Apl communicate with other processes, and the use of Apl in multitasking and possibly even multiprocessor environments, I feel that I should attempt to preach the virtues of a method that I have for a number of years found to be simple, powerful, and effective.

I look forward to hearing from you,

Yours sincerely,

J.B. Webber.

A Pipe has two ends ...

J.B. Webber. Rm. 15, Physics Lab.,
University of Kent, Canterbury.
Kent. CT2 7NR

Using Apl in a multiprocess/multiprocessor environment.

A proposal for a flexible but easy to use syntax.

Many mechanisms have been proposed for accessing data, and running processes external to Apl.

For some of these, the syntax is bizarre, for others a detailed knowledge of some other language (sometimes even assembler) is necessary to do the simplest thing.

Some, such as shared variables, offer a highly efficient method of passing data between two processes *in the same processor,* without having to copy the data.

A number of methods either allow the passing of data and/or commands to an external shell (such as the C-shell under Unix), or allow the results of commands to the shell to be imported into Apl. These in effect connect Apl to one end (but not both ends) of a pipe of shell commands.

I offer for general consideration a device that allows data to be ***piped*** *out of Apl, through (a series of) shell commands, and back into Apl.*

It works by connecting ***both ends*** *of a pipeline of shell commands to an Apl statement.*

The output of the first part of the Apl expression fills the pipe connected to the standard input of the first command in the pipeline of shell commands; the next part of the Apl expression receives the output of the pipe connected to the standard output of the the last command in the pipeline.

The commands may be executed in the same processor, or remotely, with no change in syntax.

Apl : Piping

The symbol that I use is a Quad overstruck with a vertical bar:

[|] *or :* **.pp** *or could use :* **[]pipe**

The general syntax for ***pipe*** is : ***L*** **[|]** ***R***

TextResult <- 'shell commands' [|] TextData

Pipes data ***R*** *through shell commands* ***L*** *, and back into Apl.*

Data returned will be a vector; and may be empty.
Data returned will be character, as must both ***L***, ***R***
L, ***R*** will be sent in ravel order.
(I currently require them to be vector)

If the shell command line requires no data,
then ***R*** is ignored (and may be empty)
i.e.:
ShellProgs <- 'ls *.sh ~/*.sh' [|] ''

If the shell command line returns no data,
then the data returned will be null (empty vector).
i.e.:
'cat - | sort | pr -5 | lpr' [|] []BOX []NL

an example :

```
ShellCmd <- 'fortranfit -s',(,FORMAT aplfit Data),' -n',,FORMAT 1 TAKE RHO Data
Synth <- fngen  Coeff <-  REV 8 TAKE EXECUTE ShellCmd  [|]  ,(FORMAT Data),[]L
```

This calls an existing powerful non-linear-parameter fitting routine, giving it as command line arguments both
-s : approximate starting values for the coefficients, and
-n : the number of data pairs.
The Fortran routine reads the data pairs from it's standard input, and writes the optimum coefficients (among other information) to its standard output. Apl then uses these coefficients to generate a curve for comparison.

This has the advantages that specialised or highly efficient system or pre-compiled routines (or a pipeline of such commands) can be used in the middle of a sequence of Apl operations; that the syntax of the commands is either pure traditional Apl, or pure shell commands, both of which are likely to be already known; and that it is sufficiently powerful to include within it the functionality of other more ad hoc constructs.

History of Pipe

I first implemented ***pipe*** a number of years ago (and then documented it in an internal UKC Apl manual dated 12th Jan. 83).

The amount of code needed to add ***pipe*** is quite modest (just a few pages of C); one might hope that it would be added to all versions of Apl running on systems that supports shells with pipes - even the Atari ST has Mark Williams' Msh, Beckmeyer's MT-Csh, and Poletiek's CRAFT/GPshell.

Pipe was originally added as an extension to a version of Apl under Unix that has been variously written/worked on by Ken Thompson, Ross Harvey, Douglas Lanam, ets/jrl/rww/dcw.

It has proved to be a powerful, flexible and useful tool, and is now employed in many of my routines. As well as giving these routines access to shell and pre-compiled programs, pipe allows them to make calls on languages like Maple (an algebraic manipulation program), to determine such things as the analytic differential of user supplied functions.

Some Further Points on use and syntax :

a) One could permit ***pipe*** to accept numeric data, and convert it internally to character; although I was tempted by this, I felt it wise to leave it as accepting only character, as a reminder to naive users that what comes out of ***pipe*** is to be treated as character. Numeric data usually has to be in a particular format, anyway, even when in character form.

b) With regard to the choice of passing character data only down the pipelines, I wanted a mechanism that would be independent of what is at the end of the pipe : i.e. of both the version of say Fortran/Pascal/...etc., and of the hardware it was running on. Thus each end knows how to do it's own conversion to and from character, and so does it. If one insists on preparing data inside Apl, to fit some arcane format, then one must use []DR to fool Apl into thinking that it has characters for output. It is then quite clear in the code that one is doing something non-standard.

c) It is unfortunate that the 'information flow' in Apl is right to left, but in standard shell script left to right. However one is used to both of these, and so there is little confusion. If this is felt to be of concern, shells in the newer multiprocessor operating systems do not have this restriction on their syntax, as discussed next.

Apl + Multiple Processors : Piping

> *It is with the advent of **multiprocessor systems**, and the wish to use **Apl** to easily access the power of these, that I feel it is now particularly relevant to raise this **pipe mechanism** for discussion.*

There are currently many different multiprocessor configurations under active investigation around the world; Vector and Array processors are clearly applicable, and have in the past been implemented; it is however the highly flexible Multiple Instruction Multiple Data (MIMD) configurations that currently are of great interest, (particularly with the advent of systems containing multiple 1.5 MFlop T800 floating point Transputers).

If we consider the ways that we can most easily utilise the power of these systems, answers are clearly :

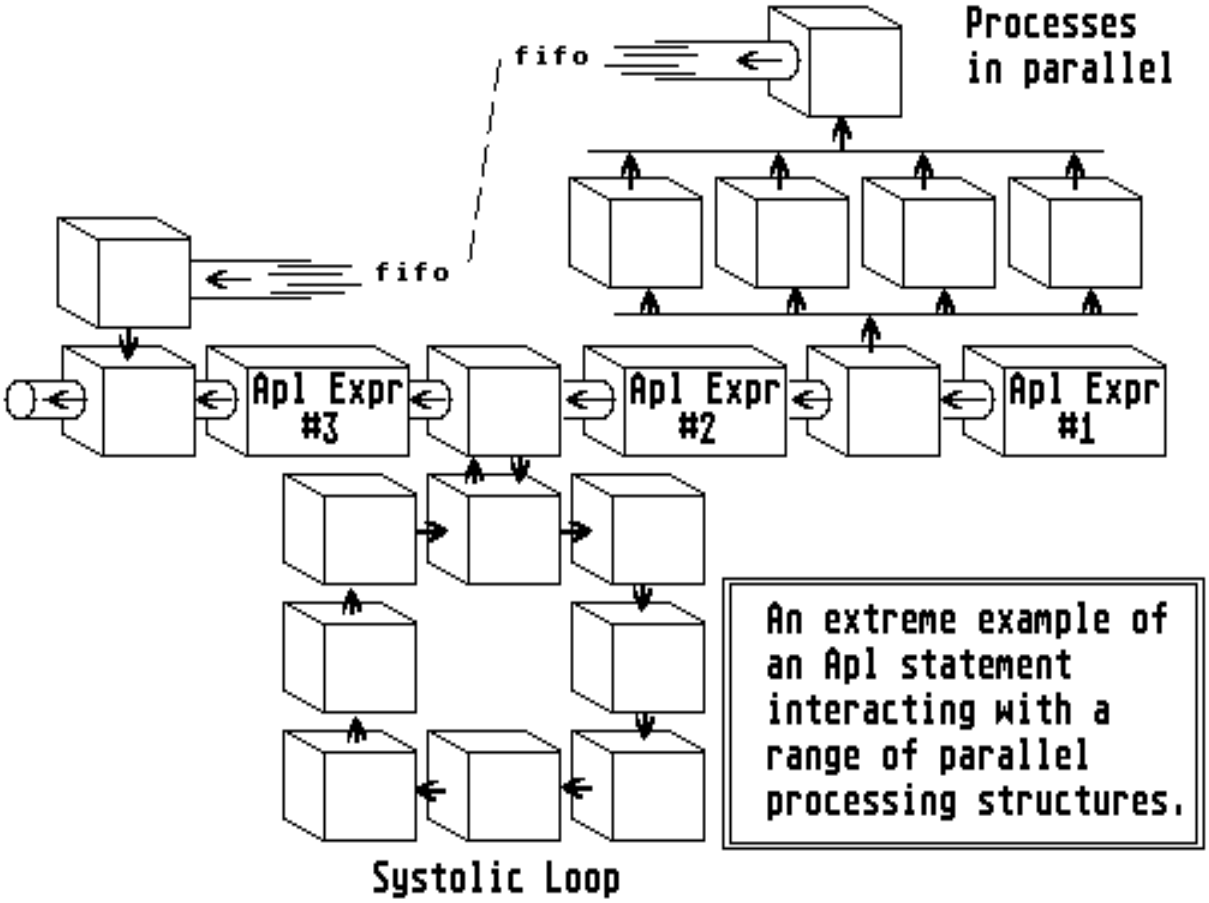
a) to attempt to apply Apl.
b) to make whatever use we can of the multitask/multiprocessor operating systems and shells that are under development for use with them.

If we examine one of these systems (Helios), we find that the concept of a pipe between processes is extended, to allow piping in either direction (i.e. to either left or right), or even bidirectionally. A further vital extension is the addition of named pipes, termed *fifos* (first in first out buffers). These allow both the easy definition of closed ring topologies, and also enable configurations where a pipe is written to at one point in a serial program, and read at a later point when the result is needed.

Thus one of the ways that we can use Apl is as components in a (complex) structure of processes, with each component farmed out by the OS to a different processor.

It is for the more complex configurations, particularly ones where the ‘main’ program is in Apl, that adding this ability to *pipe* data to and from a shell external to Apl gives us the ‘hooks’ that we need to effectively exploit the multiprocessing capability on such machines from day one. By the careful use of *fifos,* we have the option of synchronous or nonsynchronous communication and processing.

> *Other techniques may be more powerful in particular instances (such as the ability to explicitly initiate processes on remote processors) but will undoubtedly take longer to implement, and be less ‘clean’ than this straight forward piping mechanism.*


Processes
in parallel
fifo
fifo
Apl Expr
#3
Apl Expr
#2
Apl Expr
#1
An extreme example of
an Apl statement
interacting with a
range of parallel
processing structures.
Systolic Loop